Discovering the Semantic Web through join algorithms using SPARQL

Tamas Illes
Durham University

Abstract

In this paper we implemented several join algorithms in SPARQL over RDFs: inner, outer, left-outer, right-outer, full, and a slightly more complex join, to study and compare them with respect to their running times and number of returned results. In terms of correctness, all joins returned accurate results, but the optimal relation between their execution times and number of results were not found. Our experiments suggested a weak power-like relation, however further testing and inspection would be needed to call it optimal. Nevertheless we found that queries with lower number of results were executed much faster than those with high number of results.

1 Introduction

Today the Internet could provide us an unlimited amount of information, however this information may be stored in various ways which may require various access modes. Integration of all of this data would provide huge benefits for the users of the Internet who wish to combine/cross-reference this data. Using RDF, which is a general purpose language for the Web, merging data from various resources, and SPARQL, which is a query language for RDF, would allows us to process not just one data source like in SQL, but any number of them, which gives access to Linked Data for the Semantic Web. [1]

This paper provides some examples and experiments of join algorithms written in SPARQL over RDF resources using the *dbpedia* database. We measured the performance and the number of returned results for each algorithm and compared them to one another, attempting to find a trend in the execution times.

2 Methodology

As mentioned above, implemented and tested multiple different kinds of join algorithms in SPARQL over RDFs, moreover we looked at the powerful features of the Semantic Web itself. The construction of our joins in this paper were similar to those from Structured Query Language (SQL):

In our case we used the *Film* and *Book* entity types from the *dbpedia* database, since it made sense to study how they relate to one another through our choice of joins. Thus we asked the questions:

- Which books do not have corresponding films?
- Which films do not have corresponding books?
- Which cases were made into both films and books?
- Which cases were made into either films or books?

On Figure 1 A and B are equivalent to our *Film* and *Book* ontology classes, respectively. Hence our joins translate as follows:

left-outer join: films that have no corresponding books

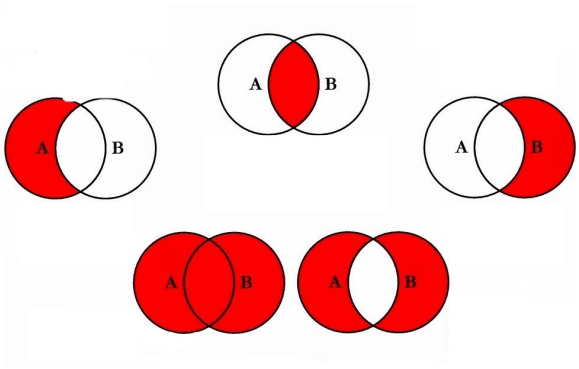

Figure 1: Joins used in our queries: left-outer (left), inner (top), right-outer (right), outer (bottom right) and full join/union (bottom left).

right-outer join: books that have no corresponding films
inner join: only films (or books) that have corresponding books (or films)
outer join: films and books that do not correspond to one another
full join: all films and all books regardless if they match up or not

To obtain these joins, we used the ".", "*MINUS*" and "*UNION*" operations of SPARQL over the $Film$ and $Book$ classes. For instance, the outer join was derived by

$$(Film \quad UNION \quad Book) \quad MINUS \quad (Film \quad . \quad Book)$$

In addition, we constructed a slightly more complicated query, which counted the number of people per nationality amongst the authors and directors of books with matching films and films with matching books, respectively. However we found that the *dbpedia* database is incomplete in the sense that even if there was both a film and a book of any particular case, it still belonged to a single entity type: either a book or a movie. The other one was just listed as an $rdf : type$, but was not accessible. Therefore for any case which was both film and book, we were only able to find either the director of the film (if it was listed under the movie entity type) or the author of the book (if it was listed under the book entity type). This could possibly be further investigated and resolved, but due to time constraints we could not achieve this, though this example still demonstrates some useful properties of joins: looking for authors/directors of books/films over multiple entity types (in our case: ontology and property types) using nested UNION features to combine triples, furthermore filtering the title of the books/films to be english, checking for the nationality (if exists) of the authors/directors and counting the results per nationality.

In section 3, to validate the correctness of the join algorithms, we checked whether or not the count of the results adds up to the right number (other than the *complex join*). In addition, to investigate their performance we measured their execution times and compared them to one another, furthermore we analysed the mean running time of each algorithms with respect of their number of results.

3 Results of the comparison and discussion

As indicated on Figure 2, the counts add up to the expected sums (on the right), thus the algorithms returned the correct number of results. Note that it did not make sense to include the *complex join* (which was slightly modified for this table to show the number of results) and that the count for the inner join had to be multiplied by two since those cases were found in both film and book classes, thus needed to be counted twice.

To measure the execution times of the algorithms, we ran the algorithms using a python wrapper ($SPARQLWrapper$) 10 times and outcome is shown on the left graph of Figure 3.

joins	complex	inner	outer	left-outer	right-outer	full	2 x inner + outer	left outer + right outer
count	164	218	166061	42981	123080	166497	166497	166061

Figure 2: Validating the number of results for various joins.

Clearly, the running times of the *complex* and *inner* joins were much lower relative to the rest, while the others (outer, left-outer, right-outer, full) had rather similar results to each another. One can notice a spike in execution time on the 3rd instance for nearly all joins, which might be due to some background CPU usage during execution or the due to slow response from *dbpedia*.

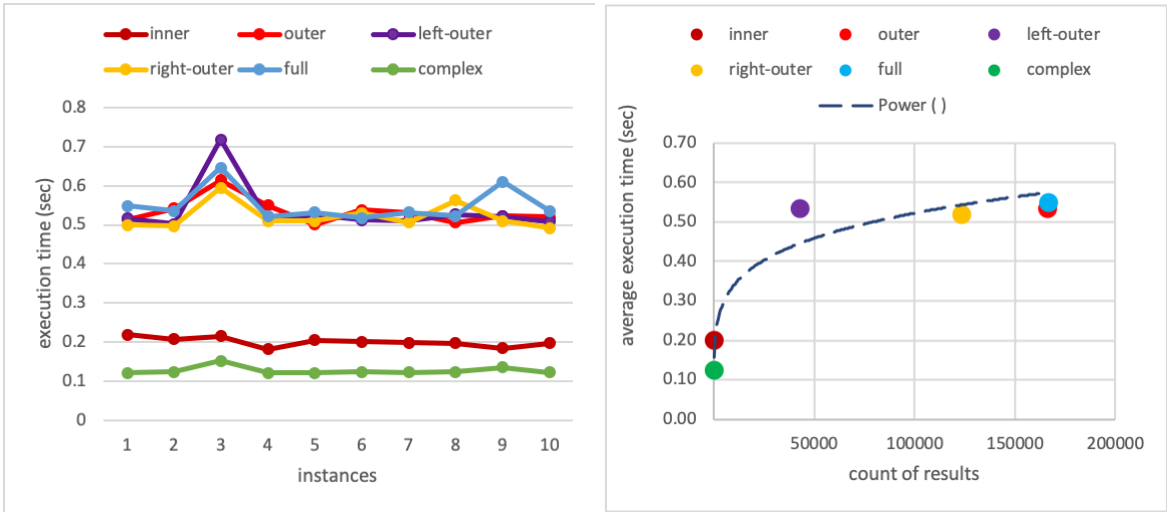

Figure 3: The execution times of joins for 10 instances on the left and the average running times with respect to the number of outcomes on the right.

Moreover, we observed that the counts from Figure 2 have a similar trend: low for *complex* and *inner* joins, and much higher for the rest. This signified some kind of relation between the counts and the execution times, thus we plotted the mean running times against the number of results on the right hand side of Figure 3. On the left diagram, the relatively big difference in running times for low number of results (*complex*, *inner*) and the small variations for much higher counts suggested a power-like relation. Indeed, we found that a power trendline fits this data the best, as one can see on the right chart of Figure 3, however this is still not an optimal since the running times in our experiments tend to shoot up to around 0.52 seconds at around 45000 counts and stay between 0.5 and 0.6 seconds for higher counts.

4 Conclusion

To summarise, we successfully implemented several join algorithms and verified their results, moreover we concluded that the running times were low for low number of counts and that they grow with the counts but once they reach 0.5-0.6 seconds they stop increasing even though the counts keep growing. Thus the suggested power-like relation did not prove to be optimal for this data, further investigation and testing is required to improve or verify this relation.

If one chooses to repeat this experiment, we could suggest to run more tests on several databases, using various entity types and joins techniques to find a perhaps more optimal relation regarding the execution times.

5 References

[1] https://www.w3.org/2007/03/VLDB/

ANNEX

```
from SPARQLWrapper import SPARQLWrapper, JSON
import time
import rdflib
import json
from statistics import mean
from matplotlib import pyplot as plt

def leftouterjoin(sparql):
	sparql.setQuery("""
		PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
		PREFIX foaf: <http://xmlns.com/foaf/0.1/>
		PREFIX dbo: <http://dbpedia.org/ontology/>
		SELECT DISTINCT str(?name) AS ?only_books, ?label AS ?link
		WHERE {
		{?label rdf:type dbo:Book} MINUS
		{?label rdf:type dbo:Film . ?label rdf:type dbo:Book} .
		?label foaf:name ?name}
		ORDER BY ?name
	""")
	sparql.setReturnFormat(JSON)
	results = sparql.query().convert()
	# print(results)
	# for result in results["results"]["bindings"]:
		# print(result["only_books"]["value"])
		# print(result["link"]["value"])
		# return (result["name"]["value"])

def rightouterjoin(sparql):
	sparql.setQuery("""
		PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
		PREFIX foaf: <http://xmlns.com/foaf/0.1/>
		PREFIX dbo: <http://dbpedia.org/ontology/>
		SELECT DISTINCT str(?name) AS ?only_films, ?label AS ?link
		WHERE {
		{?label rdf:type dbo:Film} MINUS
		{?label rdf:type dbo:Film . ?label rdf:type dbo:Book} .
		?label foaf:name ?name}
		ORDER BY ?name
	""")
	sparql.setReturnFormat(JSON)
	results = sparql.query().convert()
	# print(results)
	# for result in results["results"]["bindings"]:
	#	print(result["only_films"]["value"])
```

```
        #       print(result["link"]["value"])

def fullouterjoin(sparql):
        sparql.setQuery("""
                PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
                PREFIX foaf: <http://xmlns.com/foaf/0.1/>
                PREFIX dbo: <http://dbpedia.org/ontology/>
                SELECT DISTINCT str(?name) AS ?either_book_or_film, ?label AS ?link
                WHERE {
                { {?label rdf:type dbo:Book} UNION {?label rdf:type dbo:Film} } MINUS
                {?label rdf:type dbo:Film . ?label rdf:type dbo:Book} .
                ?label foaf:name ?name}
                ORDER BY ?name
        """)
        sparql.setReturnFormat(JSON)
        results = sparql.query().convert()
        # print(results)
        # for result in results["results"]["bindings"]:
        #       print(result["either_book_or_film"]["value"])
        #       print(result["link"]["value"])

def innerjoin(sparql):
        sparql.setQuery("""
                PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
                PREFIX foaf: <http://xmlns.com/foaf/0.1/>
                PREFIX dbo: <http://dbpedia.org/ontology/>
                SELECT DISTINCT str(?name) AS ?book_and_film, ?label AS ?link
                WHERE {
                ?label rdf:type dbo:Film . ?label rdf:type dbo:Book .
                ?label foaf:name ?name}
                ORDER BY ?name
        """)
        sparql.setReturnFormat(JSON)

        results = sparql.query().convert()
        # with open('inner.json', 'w') as outfile:
        #       json.dump(results, outfile)
        # print(type(results))
        # for result in results["results"]["bindings"]:
        #       print(result["book_and_film"]["value"])
        #       print(result["link"]["value"])
def fulljoin(sparql):
        sparql.setQuery("""
                PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
                PREFIX foaf: <http://xmlns.com/foaf/0.1/>
                PREFIX dbo: <http://dbpedia.org/ontology/>
                SELECT DISTINCT str(?name) AS ?book_and_or_film, ?label AS ?link
```

```
                WHERE {
                {?label rdf:type dbo:Film} UNION {?label rdf:type dbo:Book } .
                ?label foaf:name ?name}
                ORDER BY ?name
        """)
        sparql.setReturnFormat(JSON)
        results = sparql.query().convert()
        # print(results)
        # for result in results["results"]["bindings"]:
        #       print(result["book_and_or_film"]["value"])
        #       print(result["link"]["value"])

def complexjoin(sparql):
        sparql.setQuery("""
                PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
                PREFIX foaf: <http://xmlns.com/foaf/0.1/>
                PREFIX dbo: <http://dbpedia.org/ontology/>
                PREFIX dbp: <http://dbpedia.org/property/>


                SELECT  str(?nat) AS ?nationality, (COUNT(?nat) AS ?count)
                WHERE {
                ?label rdf:type dbo:Film , dbo:Book .
                {{?label dbo:author ?author_director} UNION {?label dbp:author
?author_director}}
                UNION
                {{?label dbo:director ?author_director} UNION {?label dbp:director
?author_director}} .
                ?label foaf:name ?name . FILTER(LANG(?name)="en") .
                ?author_director rdf:type foaf:Person .
                OPTIONAL{?author_director dbp:nationality ?nat}
                }

                ORDER BY DESC (?count)

        """)
        sparql.setReturnFormat(JSON)
        results = sparql.query().convert()
        # print(results)
        # for result in results["results"]["bindings"]:
        #       # print(result)
        #       try:
        #               print(result["nationality"]["value"])
        #               print(result["count"]["value"])
        #       except:
        #               pass
```

```
if __name__ == '__main__':
	sparql = SPARQLWrapper("http://dbpedia.org/sparql")
	innertime=[]
	outertime=[]
	lefttime=[]
	righttime=[]
	fulltime=[]
	complextime=[]
	for i in range(50):
		# print(i)
		mytime = time.time()
		inner = (innerjoin(sparql))
		innertime.append(time.time()-mytime)
		# print(i)
		mytime = time.time()
		outer = (fullouterjoin(sparql))
		outertime.append(time.time()-mytime)
		# print(i)
		mytime = time.time()
		left = (leftouterjoin(sparql))
		lefttime.append(time.time()-mytime)
		# print(i)
		mytime = time.time()
		right = (rightouterjoin(sparql))
		righttime.append(time.time()-mytime)
		# print(i)
		mytime = time.time()
		full = (fulljoin(sparql))
		fulltime.append(time.time()-mytime)
		# print(i)
		mytime = time.time()
		complex_ = complexjoin(sparql)
		complextime.append(time.time()-mytime)

	# print((left+right) == outer)
	# print((left+right+inner*2) == full)
	# print(inner)

	print("{1}\ninner join execution time: {0}\n".format(mean(innertime),innertime))
	print("{1}\nouter join execution time: {0}\n".format(mean(outertime),outertime))
	print("{1}\nleft join execution time: {0}\n".format(mean(lefttime),lefttime))
	print("{1}\nright join execution time: {0}\n".format(mean(righttime),righttime))
	print("{1}\nfull join execution time: {0}\n".format(mean(fulltime),fulltime))
	print("{1}\ncomplex join execution time:
{0}\n".format(mean(complextime),complextime))

	plt.plot(innertime)
```

```
plt.plot(outertime)
plt.plot(lefttime)
plt.plot(righttime)
plt.plot(fulltime)
plt.plot(complextime)
plt.show()
```